A Hierarchical Simulator Based on Formal Semantics

Marina Chen and Carver Mead
California Institute of Technology
Pasadena, California 91125

5068:TM:83

A preprint of the paper to appear in Proceedings of
The Third Caltech Conference on VLSI

A Hierarchical Simulator based on Formal Semantics*

Marina Chen and Carver Mead†

Simulation consists of exercising the representation of a design on a general purpose computer. It differs from programming only because the ultimate implementation will be in a different medium, say a VLSI chip. In order for simulation to be in any sense effective, the simulated system must perform the same function as the ultimate implementation. A VLSI chip is a highly concurrent object; the simulation of such a chip amounts to programming a highly concurrent system. It follows that any demonstrably correct simulation technique will be one of the two types:

(1) The entire design is represented as an implementation with objects which are abstract models of the medium at the bottom level (e.g. transistor model). The simulation operates on a representation which is a direct image of the fully instantiated implementation in the medium.

(2) The design is represented as a hierarchy of implementations. Each level of implementation is constructed of objects which are abstract models of the implementation at the level below it. The simulation operates on a hierarchical representation where each level is refined by the level below it.

The first approach requires only a model of the implementation medium. The second approach requires, in addition, a general principle for obtaining an abstract model from a given implementation of objects at the lower level. The second approach is similar to using induction as a proof technique. Instead of proving all possible cases embodied in a

*This work is sponsored by the System Development Foundation.

†Computer Science Department, California Institute of Technology, Pasadena, California.

theorem, one proves the base case, and shows the hypothesis at level n is true with the assumption that the hypothesis is true at level $n-1$. In the context of simulation and modeling, it is necessary to first establish the transistor level model, then show how to obtain a model at level n given a model at level $n-1$. The second approach yields a more efficient simulation and a clearer conceptualization of the design much the same way the inductive technique does to a proof.

In [4], we described a model of computation with formal semantics for highly concurrent systems. This method of obtaining semantics for a concurrent system is exactly the same as the principle used to obtain an abstract model for a design. In this paper, we present a hierarchical simulator which uses this composition and abstraction principle together with the formal model for MOS switch-level circuits developed by Bryant [2] as the basis.

Switch-Level Simulation

In [2], a circuit is approximated by a series of *logical conductance networks*. Transistors that are *on* are represented as a high conductance and those that are *off* as a zero conductance. The simulation of a circuit consists of obtaining the steady state solution for each conductance network and then updating the transistors whose gates have changed to obtain a new conductance network. The process continues until the topology of the conductance network itself no longer changes. The transition from one conductance network to the next is based on the unit-delay model of the switching of the transistors with respect to their gate voltage. Therefore, attaining the steady state for a transistor network involves obtaining two nested levels of steady state, the inner *conductance-level* and the outer *transistor-level*.

Multi-level and Mixed-level Simulation

In the design of a VLSI system, the traditional levels of hierarchy are circuit level, gate-level, and register transfer level. This partitioning helps designers focus on one particular level of design at any given time. When they focus on the register transfer level, for example, they can reason about the overall design in terms of the functionality of the interconnected blocks and a given timing scheme, without worrying about the details inside each block. On the other hand, when they are designing at the circuit-level, the focus is on one functional block at a time rather than the whole system. Ideally, if the overall system design is shown to be correct in terms of the functional blocks, and each functional block is shown

to be correct in terms of its circuit-level or gate-level implementation, the designers need not examine the correctness of the detailed implementation across two different functional blocks, i.e., each of these hierarchical levels provides an abstraction of the level below it. The functionality of the overall design will always be preserved when the designers cross the different levels. The complexity of a large system design can only be effectively managed through these levels of abstraction. Preserving the functionality, i.e., maintaining consistency between hierarchical levels, is the most important property of a hierarchical simulator, and the most difficult to achieve.

A multi-level simulator, when used as a tool to verify a hierarchical design, should provide a way to ensure the consistency of the design throughout all levels. On the other hand, the simulator should allow blocks of different levels to be connected through proper interfaces which handle the timing and the matching of various input/output data types. The key issue in such a multi-level simulator is the interface mechanism. In this paper, we present a simulator in which a uniform representation is used at all levels of the design. We show a method of abstraction with which the consistency can be maintained, and describe how the timing and data types can be properly interfaced.

Semantic Hierarchy and Syntactic Hierarchy

In programming languages, *macros* and *procedures* or *functions* have long been recognized as two different ways to facilitate programming. Macros are used only at a syntactic level to ease the specification, and do not provide any semantic abstraction, since they are expanded during compilation. The object code of a program using macros is exactly the same as its counterpart without using macros. However, procedures and functions are used not only to facilitate specification, but to encapsulate a piece of code with a well-defined interface to other parts of a program. Ideally, a function should not allow any side-effects and therefore provides a semantic abstraction. We make the distinction of the *syntactic hierarchy* vs. the *semantic hierarchy* in a simulator in a way analogous to the distinction between macros and procedures in a programming language. The syntactic hierarchy in the simulator serves two purposes: One is ease of specification, just as macros in a programming language; the other is that it contains information about spatial locality. Since it has been observed that activities in circuits tend to be local [2], this information can be exploited by the simulation algorithm to achieve better performance. Unlike simulators which take a flat network specification where the locality information has been thrown away by

the preprocessing to be recovered later on by topological analysis, this simulator takes advantage of user's hierarchical specification and requires neither preprocessing nor topological analysis.

Syntactic Cells. In the context of a switch-level simulator, transistors and nodes are objects from which everything else is constructed. They are bottom level cells of the semantic hierarchy. We call this bottom level the *conductance-level* since an active device (transistor) is approximated by a passive conductance. A *syntactic cell* is a circuit consisting of an interconnection of transistors and nodes where there is no restriction on the input nodes or the output nodes of the cell. The state of a node in a syntactic cell can directly or indirectly influence nodes in the others and thus the cell provides no abstraction for the behavior of a circuit.

Figure 1 shows an nMOS *exclusive nor* cell which is an example of a syntactic cell.

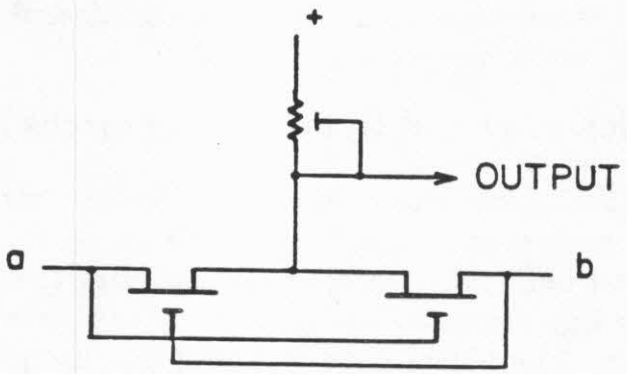

Figure 1. A non-restoring exclusive nor circuit XNOR.

Although a syntactic cell like the one shown in Figure 1 is a composition of transistors and nodes, it is at the same semantic level (or of the same semantic type) as transistors and nodes. A syntactic cell can also be a composition of other syntactic cells of the same semantic type, although these cells can be nested at an arbitrary depth in the syntactic hierarchy. No separation of the hierarchy into leaf cells and composition cells [10] is required, i.e., a transistor or a node can be composed with a syntactic cell directly without making it into a syntactic cell by itself. The simulation of such a cell produces exactly the same result as the circuit represented without hierarchy. The equivalence of Bryant's model of a flat network with our hierarchical represented network can be shown by straightforward induction on the level of the syntactic hierarchy.

The semantic hierarchy is constructed for abstracting the behavior of circuits. A syntactic cell is made into a semantic cell if an abstraction of its behavior is desired. This new *primitive* semantic cell is used as an "atom" in a new syntactic hierarchy which in turn is used within each semantic level in order to clarify the specification and express the locality of cells.

For simulator based on a switch-level model, the semantic levels can be the bottom level transistors and nodes (conductance-level), gate-level, clocked cell level (in a more general sense including a self-timed [11] module with request and acknowledge signals), register transfer level, and other higher levels. We call all levels at and above the clocked cell level the *functional level* since the functionality of cells at those levels can be abstracted.

Gate-level Cells. A *gate-level* cell is a composition of conductance-level cells (transistors, nodes and syntactic cells composed of them) with the restriction that the input nodes be uni-directional, i.e., the input nodes are gates of transistors. Figure 2 shows a gate-level cell. Note that the XNOR circuit in Figure 1 is not a gate-level cell.

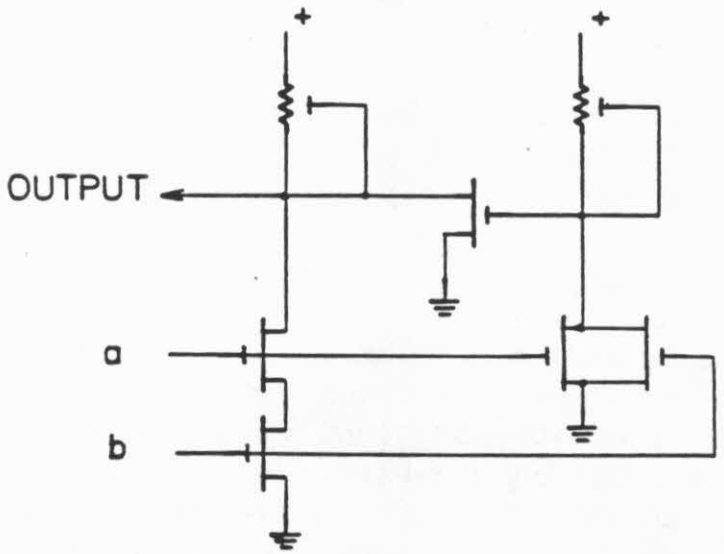

Figure 2. A restoring exclusive or gate XOR.

According to the unit-delay model of a transistor, the gate node of a transistor will not affect the state of the transistor until a given

conductance network is settled. Since each of the input nodes of a gate-level cell is the gate node of a transistor, no intermediate state on that node will be seen by the cell until the node reaches the inner conductance-level steady state described above. The gate-level cell is affected by each of its inputs when each of these gate nodes reaches its steady state and causes the corresponding transistor to change state. Notice that to abstract the behavior of a gate-level cell, we can only discard the intermediate states on all of its output nodes (prior to its reaching the inner conductance-level steady state). We cannot just keep the outer transistor-level steady state and discard all the conductance level steady states. Therefore, the conventional way of thinking about a gate-level cell as a functional block (in which all intermediate states before reaching the outer transistor-level steady-state are discarded) is not formally correct. This incorrect abstraction has to be remedied by some other analysis in the design process. For a simulator, such incorrect abstraction will miss, for example, the glitch in the circuit shown in Figure 3.

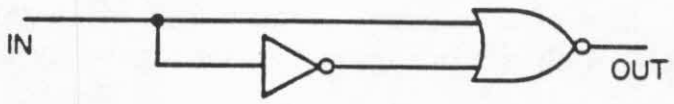

Figure 3. A gate-level circuit that generates logic 0

If we reason about the circuit shown in Figure 3 at the functional level, the output of this cell will be a logic 0 for all possible inputs. Yet when the input is initially 1 and switches to 0, the output will become a logic 1 before it settles back to 0. If the inner conductance-level steady state is kept, the simulation output will reflect the glitch properly. In actual design practice, one often reasons about interconnections of gate-level cells which behave like the above example, even though the functional abstraction is not formally correct. This formally incorrect but practically valuable abstraction works because one adopts a timing discipline in composing gate-level cells. The timing discipline ensures that each combinational circuit in a given network can only be affected by the steady state value of the output of the combinational cell to which it is connected. Familiar examples of this discipline are a two phase non-overlapping clock scheme where the clock period of each phase is long enough for a combinational circuit to settle, and a self-timed request-acknowledge signaling. An intermediate value on an output node will not propagate because of the timing discipline in the same way that the

intermediate value of a local variable will not be returned by a function in a programming language. Therefore the timing discipline provides a semantic abstraction similar to a function in a programming language.

Clocked Cells. A clocked-cell is a composition of conductance level cells with the restriction that all input nodes must satisfy a timing discipline which insures that only the steady-state of the output nodes can be seen by other cells to which they are connected. (Figure 4 shows a clocked cell formed by two other clocked cells.)

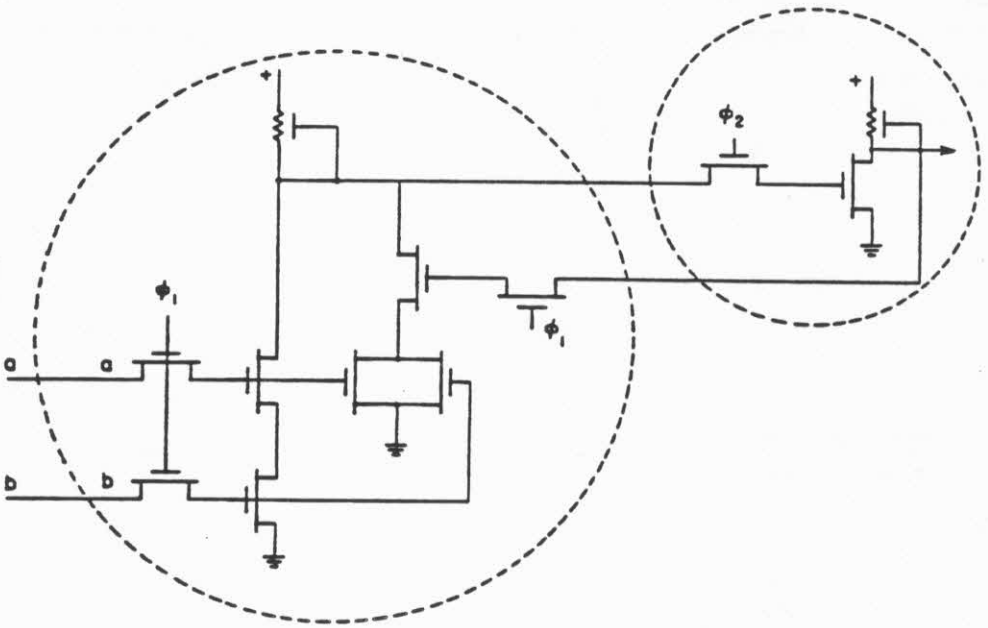

Figure 4. A clocked Muller C-element containing two sub-cells

The steady state of a primitive clocked cell (which does not contain any other clocked cell as a component, for example, as each of the sub-cells shown in Figure 4) is the steady state at transistor-level for that cell. Each of these cells can only see the steady state of other cells and therefore provides an abstraction. The gate-level abstraction is at a lower level than the clocked cell level since not only the steady state of the transistor-level is kept but all the steady states at the conductance level are kept as well. Bryant [2] points out that the gate-level cells provide a useful modeling abstraction since it is not necessary to keep track of the *signal strength* used in the conductance-level. Experience

with MOS design has shown that specifying a chip in terms of gate-levels cells is not convenient because the restriction on inputs does not allow effective use of pass transistors. Since clocked cells are usually small, we use them as the semantic level immediately above the conductance-level cells. Sub-circuits within them are represented as syntactic cells. In this simulator we therefore do not support the gate-level abstraction. In a technology other than MOS, the gate-level may well be an appropriate level of abstraction, and is very easy to implement within our simulator.

We will now proceed to illustrate the model and its use in specification and simulation by way of a simple example — a pipelined inner product element similar to that described in [8]. Figure 5 shows an example of a hierarchical partition of a single bit inner product cell IPB described in [12] which will be used later in the pipelined inner product element.

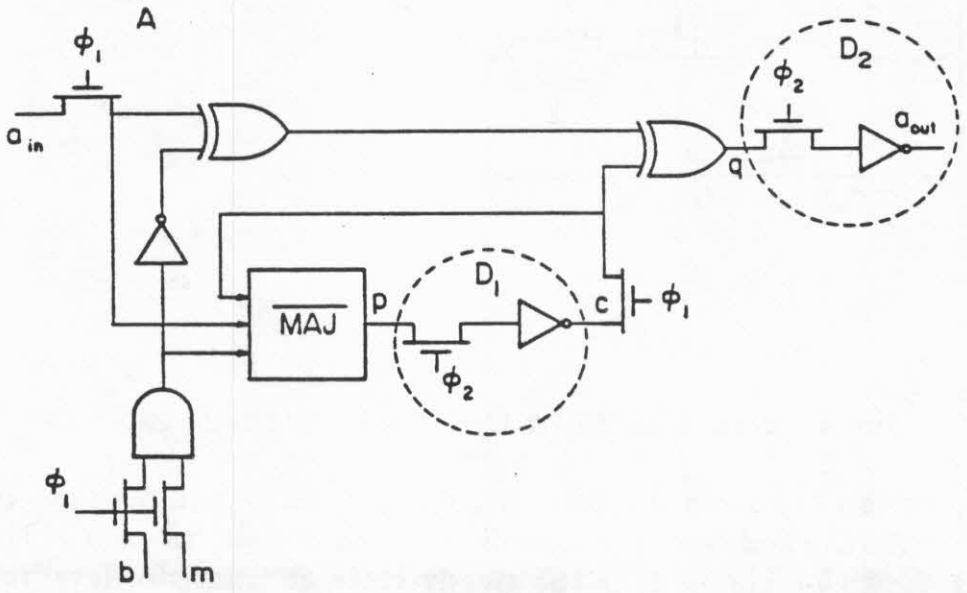

Figure 5. A single bit inner product element IPB

This cell is of type clocked-cell and consist of three clocked-cells, A which is not circled in the Figure, D_1 and D_2 which are circled. Cell A contains five syntactic cells, $\overline{MAJ}$, two XNOR gates, an AND gate, and an inverter. It also contains four transistors and four gate nodes each

with ϕ_1 on it, four input nodes a_{in}, b, m and, c and finally, two output nodes p and q.

Cells D_1 and D_2 are identical. Each consists of one syntactic cell, the inverter, a transistor and a gate node with ϕ_2 on it, an input node p for D_1 and q for D_2, respectively and finally an output node c for D_1 and a_{out} for D_2, respectively. The inputs to the IPB are a_{in}, b, m and the clocks. The output is a_{out}. There is one bit of internal state in this cell, namely the carry c. The current state of c is denoted by c_{cur} and next state by c_{next}.

The simulation of this syntactic cell proceeds by (1) obtaining the steady state of each clocked-cell (A, D_1 and D_2) using its inputs and state, independently of the other cells and, (2) transferring the outputs of one cell to the inputs of the others. Notice that a IPB cell is a syntactic cell, and therefore each of the three sub-cells is only invoked once. The IPB cell can be further abstracted to be a bit-level cell. However, it is then necessary to obtain the steady state of the IPB cell. The above procedure is iterated until the outputs of all three sub-cells do not change any more. This iteration yields the steady state of the IPB cell at the bit-level. This example shows that each semantic level requires an iteration to obtain its steady state. In the example of constructing an n bit pipelined element below, we do not use the bit-level abstraction, and so the above described procedure is invoked only once.

To verify the correctness of this circuit, the three clocked-cells are replaced by their functional specification, shown in Figure 6. The cell specified in a functional form is simulated and compared with the one (with detailed implementation) described above.

```
IF φ1 THEN
BEGIN
      q ← (a_in ⊕ (NOT(b AND m))) ⊕ c_cur ,
      p ← NOT MAJ(a_in, b AND m, c_cur);
END ELSE
IF φ2 THEN
BEGIN
      a_out ← NOT q,        c_next ← NOT p;
END;
```

Figure 6. The functional specification of a 1-bit inner product element

Notice that the data type in the functional specification is *boolean* rather than *signal* (which has two components, the signal strength and

the signal state) used in the conductance-level representation, and that the algebra of signals [2] is different from ordinary boolean algebra. In many cases, such as this inner product cell, a correct signal on the output of each clocked cell can only be a logic 0 or a logic 1 which falls into the domain of boolean algebra, although the internal nodes or even the intermediate state of the outputs can be in state X. The algebra of signals contains the signal state X, which represents an intermediate voltage between 1 and 0. The steady state output of a proper clocked cell will always be a binary value. If the output of a clocked-cell is an X, then either an error has occurred in the implementation or functional abstraction at the clocked cell level should not be applied. It is possible, in fact, very common, for the output of a clocked cell to be *unknown*, for example, when the output is a function of some state variable which has not been initialized. At every level we use the symbol $\perp$ (pronounced *bottom*) to represent an undefined value. It must not be confused with X which represents a voltage between logic 0 and logic 1 at the conductance-level. Since an intended output will never depend on an unknown value $\perp$, a possible way of handling a $\perp$ in the input of a function would be to define the corresponding output also to be $\perp$. An algebra extended this way is called *naturally extended* [9] (although there exists other possible extensions, the natural extension is used in this simulator).

Notice that even at the level of a simple 1-bit cell, we can see the kind of data abstraction that always accompanies functional abstraction. Each level has its own algebra for manipulating data and functions. A formal treatment of the model of computation that allows such abstractions (input/output mapping functions for data abstraction and fixed-point semantics for functional abstraction) is given in [4].

Word-level Cells. The functional specification given in Figure 6 is used in the next level of composition, in this example the pipelined inner product element. The performance of the simulation using this specification will be drastically improved in comparison with the circuit-level specification. Although exhaustive checking is possible for verifying the consistency between the two specifications for a single cell like IPB, it becomes very rapidly impractical as the size of the cell grows. Then it becomes necessary for the notation or language in which the design is described to have formal semantics in order to allow verification of the consistency between two different levels of specification. Space-time recursion equations described in [4] are an example of such a notation. Although the current simulator is implemented as an embedded language in an ordinary programming language, the primitives for specifying the design are a direct mapping of the above formal notation.

An n-bit pipelined inner product element can be composed by connecting n IPB cells serially (the specific scheme is shown in [12]). We call this cell IPE. It can be viewed as a word-level cell where a pulse input *lsb* indicates the start of a word (say, least significant bit of an n bit word). We can adapt the interface of this cell so that the clocks can be hidden inside the cell and the bit serial input and output can be abstracted as words. Figure 7 shows the data abstraction which maps input words $\hat{a}_{in}$, $\hat{b}$ and $\hat{m}$ into series of bits and collects output bits to be the word $\hat{a}_{out}$. The bits $\hat{a}_{in}[i]$, $\hat{b}[i]$, and $\hat{m}[i]$ are put in one by one to the IPE at its input ports $ipe_{a_{in}}$, ipe_b, and ipe_m. A suitable clock is also generated for the cell. Once the inputs and the clocks are valid, IPE is called upon to compute its result (written as IPE.*compute* in Figure 7). It computes by invoking each of the IPB cells (which invokes once each of its sub-cells A, D_1 and D_2 as described above), transfers the results of one IPB to the next and repeats until each of the result returned by all IPB cells becomes steady.

The functional abstraction of the IPE is shown in Figure 8. Again, the consistency of the two different levels can be verified by the combination of formal verification and simulating both specifications.

```
IF lsb THEN
FOR i := 0 TO n − 1 DO (loading phase)
BEGIN
      ipe_{a_in} ← â_in[i], ipe_b ← b̂[i], ipe_m ← m̂[i]; (input mapping)
      φ1 ←high, φ2 ←low; IPE.compute;
      φ1 ←low, φ2 ←high; IPE.compute;
END;
FOR i := 0 TO n − 1 DO (unloading phase)
BEGIN
      φ1 ←high, φ2 ←low; IPE.compute;
      φ1 ←low, φ2 ←high; IPE.compute;
      â_out[i] ← ipe_{a_out}; (output mapping)
END;
```

Figure 7. Interface of an n bit-serial element to higher-level specification

$$\texttt{IF } lsb \texttt{ THEN } \hat{a}_{out} \leftarrow \hat{a}_{in} + \hat{b} \times \hat{m}$$

Figure 8. The functional specification of the inner product element

The n bit inner product element can be used to construct, for example, a systolic array performing matrix multiplication [6]. At the level

of a systolic array, the inner product element is used in the functional form as in Figure 8 regardless of its implementation as bit serial or word parallel. We have to be careful however, since the mapping functions that constitute the data abstraction are never unique. The mapping may be from n serial bits to a word or from n parallel bits to a word. The mapping in the former is from time domain to an abstract word and the latter is from space domain to an abstract word. In connecting two inner product elements at this level, one needs to make sure that the output mapping function of one element is the inverse of the input mapping function of the element to which it is connected. Two IPE elements as shown in Figure 7 can be connected since the output mapping of one (from bits $ipe_{a_{out}}[i]$ for $0 \leq i < n$ to an n bit word $\hat{a}_{out}$) is the inverse of the input mapping of the other (from n bit word $\hat{a}_{in}$ to bits $ipe_{a_{in}}[i]$ for $0 \leq i < n$). The lower level has the same interface problem in a different form. It is the timing discipline used in the design that allows abstraction to the clocked cell level. This discipline can be one of several kinds. For example it may be two, three or four phase clock, or a two or four cycle request-acknowledge protocol. Once again the condition imposed on the output of one cell must match that of the input of the cell to which it connects.

The Hierarchical Design Method

The systolic array can also be abstracted from its implementation and used as an abstract machine performing matrix multiplication. The detailed procedure of abstracting a systolic array is presented in [4]. Notice that the abstraction mechanism is precisely the same at all levels. Considering the bottom-up approach, we summarize the two essential steps in the hierarchical design method.

(1) Use a cell defined by its implementation in the context of cells defined by their functional specifications. In order to adapt the interface between the cell and its context, a new cell containing the following three parts is constructed:

   i) A function which maps inputs in the data representation of the high level to the inputs at the lower level.

   ii) The implementation at the lower level.

   iii) A function which maps the outputs at the lower level to outputs in the data representation at the higher level.

This cell is now a proper cell with exactly the same interface as a higher level cell. We can use it as such or

(2) Replace such an implementation with its functional specification. These two versions shall be verified to be consistent either by simulating them against each other, by formal verification or by a combination of the two. Once the results are identical, the first cell can be replaced by its functional equivalent.

The order of these two steps can be reversed for the top-down approach. In the top-down approach, instead of an implementation of lower level functions being abstracted to be a higher level function, a higher level function is implemented by some lower level functions. Such refinement of design can be carried out until the implementation at the bottom is completed.

What we have shown is a method for constructing systems from switch-level circuits to functional blocks through successive semantic abstractions or, stated the other way around, starting with functional specification at the top level and successively refining the specification until it is implemented in the bottom level representation of the medium (in this case, switch-level model of transistors and nodes). In contrast to the conventional view of fixed hierarchical levels, the partitioning of a semantic hierarchy is flexible and problem oriented. Designers can partition each system in the way that is most natural to the design, rather than fit into rigid pre-defined levels which are not necessarily appropriate.

Implementation of the Simulator

The example shown above is very specific, and one can obviously write a multi-level simulator for this particular system. Such a simulator would be of limited use since it handles only designs partitioned in the same way. A general purpose simulator which must support arbitrary levels and mixed-level seems at first unrealistic. We approach this problem by:

1. Separating out the part that is universal to all system levels

2. Using the power of an embedded language [7].

Embedded languages. It has been observed in integrated circuit layout languages that an embedded language — a language supporting graphics primitives in an existing programming language — has the

generality and flexibility in the specification of designs that an interactive graphic layout system usually lacks. The effort of making a graphic system as powerful as an embedded language is essentially that of supporting a general purpose programming language. It is much more sensible to let the compiler of an existing language do the work. The same philosophy applies to a specification language for simulation. We build into a programming language the simulation algorithm and an interactive user interface (corresponding to the debugger in a programming environment) for testing the design. One specifies cells in an embedded simulation language by invoking primitives for transistors, nodes, syntactic cells and semantic cells. These primitives are pre-defined in the language. With the power of a general programming language, users can then specify functional abstractions, the data abstractions, and various data types at any level according to their conceptualization of the design.

Representation of Cells and the Dynamics of the Simulation. Cells are represented as *modules* in a programming language supporting separately compilable modules. The language we have used for this specific implementation is Mainsail. A cell has in it the specification of its constituents, i.e., the implementation in terms of lower level cells, and the procedure for computing its result. The former is supplied by user and the latter is incorporated automatically. A semantic cell computes by causing each of its constituents to compute until the steady state is reached; syntactic cells only step through each of the constituents once. Transistors, nodes and cells with functional specification that models their behavior compute directly. Composition cells (cells composed of sub-components) cause their constituents to compute in a recursive manner until one of the primitive functions or abstracted function is encountered. Each of these different ways of computing a cell is made into a template and the template is compiled together with the specification of its construction in the case of a composition cell. A module is typed according to the template incorporated into it. The net effect is that each module contains the functions necessary to compute its own behavior.

We use a module instead of a procedure to represent each cell because a cell can be mapped directly into a module which has a data section to represent the cell's inputs, outputs and states, and a procedure section for the specification of its constituents and computation of its behavior. Modules also have one bit of state which records whether the inputs are the same as those of the last invocation. No computation is necessary if they are the same since the outputs of the last computation are still valid. This "change bit" allows a truly efficient implementation.

Representation of Connections. The computational aspects of a system have attracted much more attention than the communication aspects partly because in a von Neumann architecture, the cost of accessing the random access memory is always the same and partly because historically, variable names cost nothing and furthermore, substitution or elimination of redundant variables has attracted relatively little attention in mathematics. Since the transfer of information does cost energy and resources (wires), it must be taken intc account in any model of a computational system. We represent the communication among modules also as a module which contains the connectivity information and various ways to transfer information, such as a uni-directional connection or a bi-directional connection.

The Universal Fixed-Point Algorithm. By viewing a computational system as an ensemble of cells and connections, we devise a fixed-point algorithm to find the steady state of a cell. The fixed-point algorithm performs a task similar to the so-called "relaxation" algorithm. It takes the implementation of a cell in terms of connections and cells in a bipartite format. The algorithm invokes each cell, i.e., causes each cell to compute independently, and after all have been invoked, then invokes each connection to transfer the data and thereby bring about the interactions among connected cells. This procedure is iterated until the steady state is reached. The bipartite arrangement of cells and connections results in the property that the order in which each cell is invoked is immaterial in the algorithm. This algorithm is shown in [3] to yield the steady state of a system. Since the above model is uniformly applied to all semantic levels, and all cells and connections are represented uniformly, only a single universal fixed-point algorithm is necessary.

Elements of a Multi-level Simulator. The following primitives, embedded into Mainsail, serve as templates for user defined circuits:

1. Cells of various types: transistors, syntactic cells, conductance networks, clocked cells, and functional cells.

2. Connections of various types: nodes, bi-directional connections for syntactic cells below clocked cell level, conductance network formation, and uni-directional connections for functional cells. The transition from one conductance network to the next is also represented as a connection.

In the object-oriented view of computation such as Simula [1], Smalltalk [5], etc., these templates are the superclasses of the user defined

classes (cells). These templates are the only structure we build into the simulator. Instead of building and maintaining data structures that represent a design, the structure is embedded in user's specification of interconnected modules. Hence, no global simulation algorithm is necessary to traverse the data structure.

Conclusions

We have described a multi-level simulator which allows user-defined levels instead of rigidly pre-defined levels. We draw a clear distinction between the modularization for ease of specification and for semantic abstraction — the syntactic hierarchy and the semantic hierarchy, respectively. An example of multi-level simulation is given which spans from circuit-level up to the abstract function of an inner product element.

With a formal model as a basis, the implementation of the simulator is simple and uniform at all levels. A single universal fixed-point algorithm is used. This approach raises the activity of simulation from a low level corresponding to macro assembly level in a programming language to a hierarchical specification corresponding directly to the conceptualization of user's design. We show how functional abstraction and data abstraction (interfaces between two different levels) can be made. These abstractions are the key to the consistency and efficiency of a multi-level simulator.

We demonstrate the importance of the formal semantics which allows functional abstraction and thereby enables an efficient simulation methodology. In simulation, showing that the specification and the implementation are equivalent is not merely desirable but absolutely essential. We hope that this working example has shown that formal semantics is an essential feature of any design tool as well as any concurrent programming language.

Acknowledgments

We wish to thank Randy Bryant for insightful discussions and suggestions on the subject of simulation, and Alain Martin for his valuable comments in the preparation of this paper.

References

[1] Birtwhistle, G. M., Dahl, O-J, Myhrhaug, B., and Nygaard, K., *Simula Begin*, Petrocelli, New York, 1973.

[2] Bryant, R.E., *A Switch-level Simulation Model for Integrated Logic Circuits,* Massachusetts Institute of Technology, Cambridge Massachusetts, March, 1981.

[3] Chen, M.C., Doctoral Dissertation, Computer Science Department, California Institute of Technology, 1983.

[4] Chen, M.C. and Mead C.A., *Concurrent Algorithms as Space-time Recursion Equations,* Proceedings of USC Workshop on VLSI and Modern Signal Processing, pp. 31-52, November, 1982.

[5] Ingalls, D., *The Smalltalk 76 Programming System: Design and Implementation,* Proceedings of the Fifth ACM Conference on Principles of Programming Systems, pp. 9-16, January 1978.

[6] Kung,H.T. and Leiserson C.E., *Algorithms for VLSI Processor Arrays,* in C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980, chapter 8.3.

[7] Locanthi, B., *LAP: A Simula Package for IC Layout,* Caltech SSP Report #1862, California Institute of Technology, 1978

[8] Lyon R. F., Two's Complement Pipeline Multipliers, *IEEE Trans. on Communications* COM-24, pp. 418-425, 1976.

[9] Manna, Z., *Mathematical Theory of Computation,* McGraw-Hill, New York, 1974.

[10] Rowson, J. A., *Understanding Hierarchical Design,* Doctoral Dissertation, California Institute of Technology, April, 1980.

[11] Seitz, C., *System Timing,* in C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980, chapter 7.

[12] Wawrzynek J. and Lin T. M., *A bit Serial Architecture for Multiplication and Interpolation,* 5067:DF:83, Computer Science Department, California Institute of Technology, January, 1983.